FPGA Acceleration of Reference-Based Compression for Genomic Data

James Arram, Moritz Pflanzer, Thomas Kaplan, Wayne Luk
Department of Computing, Imperial College London
Email: jma11@imperial.ac.uk

***Abstract*—One of the key challenges facing genomics today is efficiently storing the massive amounts of data generated by next-generation sequencing platforms. Reference-based compression is a popular strategy for reducing the size of genomic data, whereby sequence information is encoded as a mapping to a known reference sequence. Determining the mapping is a computationally intensive problem, and is the bottleneck of most reference-based compression tools currently available. This paper presents the first FPGA acceleration of reference-based compression for genomic data. We develop a new mapping algorithm based on the FM-index search operation which includes optimisations targeting the compression ratio and speed. Our hardware design is implemented on a Maxeler MPC-X2000 node comprising 8 Altera Stratix V FPGAs. When evaluated against compression tools currently available, our tool achieves a superior compression ratio, compression time, and energy consumption for both FASTA and FASTQ formats. For example, our tool achieves a 30% higher compression ratio and is 71.9 times faster than the fastqz tool.**

I. Introduction

The exponential growth in data generated by next-generation sequencing (NGS) platforms poses a big challenge for data storage infrastructures [10]. Indeed, the latest sequencing platforms generate Terabytes of genomic data in a single run, and their throughput is expected to increase 3-5 times each year. An effective solution to this challenge is data compression, which has consequently become a major area of interest in genomics research.

FASTA and FASTQ are two popular formats for storing genomic data. These formats are based on a plain text representation of the genomic data components, namely the sequences of DNA letters, quality scores, and meta data, as shown in Figure 1. These components are typically compressed independently using strategies which best suit the data characteristics. For compressing the sequence component, a reference-based strategy is often adopted, whereby the sequences are encoded as a mapping to a known reference sequence. Given the high similarity of inter-species DNA (over 99.9% for Human DNA), this strategy typically achieves higher compression ratios than non-reference-based strategies.

Several reference-based compression tools have been implemented in software, including fastqz [2], fqzcomp [2], lw-fqzip [16], and gdc2 [5]. These tools achieve higher compression ratios than general purpose compression algorithms such as gzip and bzip2, however the improvement in compression ratio comes at the cost of the compression speed. For example, most of the tools listed would require several hours to compress the data generated by a single NGS platform run.

```
1  >chr22
2  TAGAAGTTCTCTGAGACCTAGGCTTTGTGAATCCAA
3  AGGGATCTTTTTAACGAATAAAATGAATCAGGGCCC
4  AATGGGACGTGAGGGTTCCTCAGGCCAGTAGTATGG
```

(a) FASTA format. Lines with the symbol > in the first position contain meta data, the remaining lines contain sequences. No quality scores are stored.

```
1  @HWI-EAS209:5:58:5894:21141#ATCACG/1
2  GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
3  +HWI-EAS209:5:58:5894:21141#ATCACG/1
4  IIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

(b) FASTQ format. Each record is stored over 4 lines. Lines 1 and 4 contain meta-data, line 2 contains the sequence, and line 4 contains the quality scores.

Fig. 1: Genomic data formats.

Determining the mapping to the reference sequence is typically the bottleneck: accounting for over 70% of the compression time. Given the projected growth of data generated by NGS platforms, these tools will be unable to cope with the massive amounts of genomic data available in the future.

Field Programmable Gate Arrays (FPGAs) are a promising candidate for accelerating reference-based compression: first, there are several successful works on accelerating sequence alignment [1], [7], [14], which comprises a similar mapping problem; and second, the low operational clock frequencies of FPGAs allow compact and energy efficient solutions appropriate for data centres and clinical settings. Building upon our previous work on accelerating sequence alignment: we present the first FPGA acceleration of reference-based compression for genomic data. When evaluated against compression tools currently available, our tool achieves a superior compression ratio, compression time, and energy consumption for both FASTA and FASTQ formats. Indeed, our tool is easily able to cope with the rate at which data is generated by NGS platforms, creating exciting opportunities for commercial integration. The contributions of this work include:

- A new mapping algorithm based on the FM-index search operation with algorithmic optimisations targeting compression ratio and speed.
- A hardware design for the proposed mapping algorithm. Equations are derived for modelling the performance for

978-1-4673-9091-0/15/$31.00 ©2015 IEEE

different data sets and hardware platforms.
- Performance evaluation of our reference-based compression tool implemented on a Maxeler MPC-X2000 dataflow node. Comparisons are made to other compression tools currently available.

II. Background

In this section we provide background information on reference-based compression and the FM-index search operation.

A. Referential compression

The basis of reference-based compression is encoding sequences of DNA letters as a mapping to a known reference sequence. There exist several options of how to represent the mapping. One option is as a triple $\langle pos, len, sym \rangle$, composed of the position where the match occurs (pos), the length of the match (len), and the first character following the match (sym). This option yields good compression ratios when the to-be-compressed sequences and reference sequence are highly similar and only differ by single nucleotide polymorphisms (SNPs) [15]. Figure 2 shows the mapping of two sequences using a triple representation.

To achieve high compression ratios, a widely identical reference sequence, and a good mapping onto it have to be found. To find the best mapping, the reference can be represented as a suffix tree or index structure from which the longest matching parts can be derived. In addition to considering only exact matches, there have been approaches to map reverse substrings, complements or palindromes [12].

B. FM-index

The FM-index is a full-text compressed index which supports substring searching in linear time (with respect to the substring length). The FM-index is based on the Burrows-Wheeler transform (BWT) [3], a permutation of a text generated from its Suffix Array (SA) [11].

The SA of a text R is the lexicographically sorted array of the suffixes of R, where each suffix is represented by its position in R. The SA interval (low, $high$) covers a range of indices in the SA where the suffixes have the same prefix. The pointer low gives the index in the SA where the pattern is first found as a prefix, and the pointer $high$ gives the index after the one where the pattern is last found. Figure 3a illustrates the construction of the SA for a text. In this example the SA interval for the substring `A` is (1, 4). The result of searching for a substring can be represented as an SA interval. If $low < high$, the substring occurs in the text. Conversely, if $low \geq high$, the substring does not occur.

The FM-index is built from the BWT, a transformation which generates a permutation of the symbols in a text. Each position in the BWT is computed using the relationship: $\text{BWT}_i = R[(\text{SA}_i - 1) \bmod |R|]$. Figure 3a illustrates the construction of the BWT from the SA of a text. The FM-index supports substring searching through two functions performed on the BWT. Count(s) returns the number of symbols in the BWT which are lexicographically smaller than the symbol s. Occ(s, i) returns the number of occurrences of the symbol s in the BWT from positions 0 to $i - 1$. The values of these functions are precomputed and stored as arrays, as shown in Figure 3b. To compress the size of the FM-index, the Occ array is sampled into buckets of size d. In this procedure the Occ values are stored every d positions as markers, reducing the array size by a factor of d. The Occ values omitted are reconstructed by summing the previous marker and the result of counting the occurrence of the remaining positions directly from the BWT. In addition, the search operation can be simplified by adding the corresponding Count value to each Occ marker. The Occ markers and the corresponding section of the BWT are then interleaved to form the FM-index, as shown in Figure 3c. This work uses the n-step FM-index [4],

Reference: `AATGGGACGTGAGGGTTCCTCAGGCC`	
Sequence	Mapping triples $\langle pos, len, sym \rangle$
`AATGGA`	$\langle 0, 5,$ `A`$\rangle$
`TTCCACA`	$\langle 15, 4,$ `A`$\rangle$ $\langle 20, 2,$ `_`$\rangle$

Fig. 2: Triple mapping representation. Note that _ encodes a null value.

(a) R = `BANANA$`

i	SA	suffix
0	6	`$`
1	5	`A$`
2	3	`ANA$`
3	1	`ANANA$`
4	0	`BANANA$`
5	4	`NA$`
6	2	`NANA$`

BWT = `ANNB$AA`

(b) Occ(s, i)

i	A	B	N
0	0	0	0
1	1	0	0
2	1	0	1
3	1	0	2
4	1	1	2
5	1	1	2
6	2	1	2
7	3	1	2

Count(s)

A	B	N
1	4	5

(c) FM-index ($d = 4$)

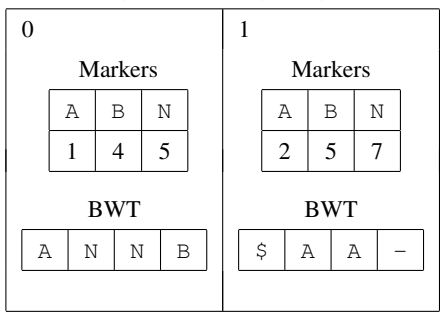

Fig. 3: Generating the FM-index. Note that `$` is the terminal symbol, the smallest symbol lexicographically.

Algorithm 1 FM-index search operation.

Input: substring Q, FM-index F with bucket size d, and the suffix array SA of the text R
Output: positions in R where Q occurs
Procedure: $\Phi(F, s, i)$ – returns Occ(s, i) from FM-index bucket F

```
1: low ← 0                                   ▷ initialise suffix array interval
2: high ← max(Occ_i)
3: for i ← |Q| − 1 to 0 do                   ▷ update suffix array interval
4:     low ← Φ(F[low/d], Q_i, low)
5:     high ← Φ(F[high/d], Q_i, high)
6:     if low ≥ high then                    ▷ terminate if symbol not matched
7:         end
8: end for
9: for i ← low to high − 1 do                ▷ get reference positions
10:    Pos ← SA_i
11: end for
12: procedure Φ(F, s, i)                     ▷ get Occ(s, i) from F bucket
13:    marker ← (F → Markers[s])             ▷ get marker value
14:    count ← 0
15:    for j ← 0 to j < i mod d do           ▷ count from BWT
16:        if s = (F → BWT_j) then
17:            cnt ← cnt + 1
18:    return marker + count
19: end procedure
```

an algorithmic modification which allows the SA interval to be updated for n symbols in each iteration of the search operation; consequently, the total number of memory accesses to the FM-index is reduced by a factor of n.

The FM-index search operation is described in Algorithm 1. Concisely written, the SA interval (low, $high$) is first initialised to the minimum and maximum indices of the Occ array. low and $high$ are then updated for each symbol in the substring (from the last to the first) using the equations in lines 4 and 5. After the final iteration, the SA interval gives the range of indices in the suffix array in which the suffixes have the substring as prefix. These indices are subsequently converted into positions in the text. Figure 4 illustrates an example of the FM-index search operation.

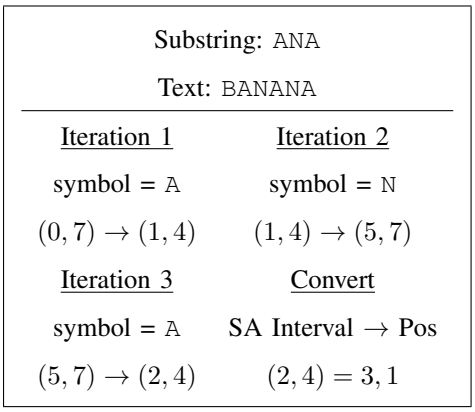

Fig. 4: FM-index search operation example.

III. Related Work

The popularity of reference-based compression has steadily increased since more complete genomes have become available and can be used as reference sequences. Several reference-based compression tools have been implemented in software, including gdc2 [5] for the FASTA format, and fastqz [2], fqzcomp [2], and lw-fqzip [16] for the FASTQ format. These tools work on similar principles: first the sequences are encoded as a mapping to a reference sequence (first order compression), then the mapping representation is compressed further using a general purpose compression tool (second order compression). The mapping is typically performed using a hash table, whereby the sequence is split into seeds which are used as keys to a hash table of the reference sequence. For most of the reference-based tools listed, the mapping process is the bottleneck. This is especially the case when large reference sequences are used due to the initialisation costs associated with hash table construction and collision handling.

To our knowledge there are no previous works on accelerating reference-based compression using FPGAs. However, there are several works on accelerating sequence alignment, which comprises a similar mapping problem. The main difference is that for reference-based compression, the sequences can be split into variable-length substrings to find the best mappings, whereas for sequence alignment the best mapping is found for the entire read. Notable works on accelerating sequence alignment using FPGAs include [1] and [7], which accelerate the FM-index search operation, and [14], which accelerates the Smith-Waterman algorithm.

IV. Mapping Algorithm

In this section we present a new mapping algorithm based on the FM-index search operation which includes algorithmic optimisations targeting compression ratio and speed.

A. Algorithm overview

In the proposed algorithm, each sequence mapping is represented by a set of triples $\langle pos, len, sym \rangle$, composed of the position where the match occurs (pos), the length of the match (len), and the first character following the match (sym). pos is stored using 4 bytes, allowing the whole Human genome to be used as a reference sequence; len is stored using 1 byte; and sym is stored using 1 byte.

The to-be-compressed sequence is mapped to the reference sequence using the FM-index search operation, as shown in Algorithm 2. Concisely written, each symbol in the sequence is matched to the reference sequence using a backward search. If a mismatch is found ($low \geq high$), or all symbols have been matched, a triple is added to the set. If any symbols remain unmatched after a triple is added, the search operation is restarted from the next symbol, where the values of low and $high$ are reset to the minimum and maximum indices of the Occ array. To decompress the sequence, the set of triples is traversed from right to left and each triple is replaced by the corresponding section of the reference sequence. The

advantages of the FM-index search operation over hash table-based approaches are:

1) The mapping performance is independent of the reference sequence length; consequently, the compression speed is not reduced when the full Human genome (3 billion symbols) is used as a reference sequence.
2) Given that genomes are rarely updated, the FM-index would only need to be computed once for use in a large number of compression jobs. This is in contrast to hash table-based approaches where there are large initialisation costs associated with table construction and collision handling.

B. Compression speed optimisation

In the FM-index search operation, two memory accesses ($F[low/d]$ and $F[high/d]$) are required to update the SA interval for each symbol; consequently, $2L$ memory accesses are required to map a sequence of L symbols. Due to its large size, the FM-index is stored in off-chip DRAM. Given that the access latency to off-chip DRAM is in the order of hundreds of cycles, and the access pattern is random, the performance of the search operation is memory-bound.

To reduce the number of memory accesses, we utilise the n-step FM-index [4]. This is an algorithmic modification to the FM-index structure which allows the SA interval to be updated for n symbols in each iteration of the search operation; consequently, the number of memory accesses is reduced from $2L$ to $2L/n$. In this work we further reduce the number of memory accesses by developing a new algorithmic optimisation which we refer to as index oversampling.

Algorithm 2 Mapping algorithm.

Input: sequence Q, n-step FM-index F with bucket size d and step size n, and the suffix array SA of the text R
Output: Set of triples $trip$
Procedure: appendTriple(pos, len, sym) appends a triple to the set
Procedure: merge($s_1, ..., s_n$) merge symbols from left to right

```
low ← 0                                          ▷ initialise suffix array interval
high ← max(Occ_i)
len ← 0
for i ← |Q| − 1 to 0 step −n do                  ▷ update suffix array interval
    s ← merge(Q_{i−n+1}, ..., Q_i)
    low_t ← Φ(F[low/d], s, low)
    high_t ← Φ(F[high/d], s, high)
    if low_t ≥ high_t then                       ▷ mismatch found: add triple
        trip.appendTriple(SA[low], len, s)
        len ← 0                                  ▷ reset mapping state
        low ← 0
        high ← max(Occ_i)
    else
        low ← low_t
        high ← high_t
        len ← len + n
    if i = 0 then                                ▷ if all symbols matched: add triple
        trip.appendTriple(SA[low], len, _)
end for
```

Algorithm 3 SA interval update for the oversampled n-step FM-index.

Input: substring Q, oversampled n-step FM-index F with bucket size d, step size n, and oversampling factor f

```
for i ← |Q| − 1 to 0 step -n do                  ▷ update suffix array interval
    s ← merge(Q_{i−n+1}, ..., Q_i)
    low ← Φ(F[low/d], s, low)
    if high − low ≥ d/f then                     ▷ point to different buckets
        high ← Φ(F[high/d], s, high)
    else                                         ▷ point to same bucket
        high ← Φ(F[low/d], s, high)
end for
```

With each update of the SA interval, the values of low and $high$ converge. After several iterations it is often the case that low and $high$ are sufficiently close that $F[low/d]$ and $F[high/d]$ point to the same index bucket. In this case only one memory access is required to update both low and $high$. If $F[low/d]$ and $F[high/d]$ always point to the same index bucket after X symbols have been aligned, then the total number of memory accesses is reduced from $2L$ to $2X + (L - X)$. The value of X is dependent on the size of the reference sequence. Tests using the full Human-genome as a reference sequence indicate that the average value for X is 13, therefore for a sequence of 100 symbols, the number of memory accesses is reduced by 1.8 times.

To eliminate cases where the values of low and $high$ are sufficiently close, but $F[low/d]$ and $F[high/d]$ point to adjacent buckets, the index is oversampled by a factor of f. In this procedure, the Occ values are stored every d/f symbols, however the BWT size remains d symbols. The trade-off is that the index size increases by a factor of f, however this can be mitigated by increasing the bucket size. If $high - low < d/f$, then $F[low/d]$ and $F[high/d]$ will point to the same index bucket; consequently, only one memory access is required to update the suffix array interval for each of the remaining symbols. The SA interval update is modified according to Algorithm 3.

C. Compression ratio optimisation

To improve the achievable compression ratio, optimisations are developed which exploit the sequence characteristics of the FASTA and FASTQ formats.

FASTA format.

The FASTA format is typically used to store pre-assembled genomes such as chromosomes and proteins. The sequences are therefore in order relative to where they appear in the genome. In our compression algorithm each line in the input file is treated as a separate sequence, therefore the highest achievable compression ratio is $L/6$, where each sequence of L symbols is encoded using a single triple of 6 bytes. To improve the compression ratio we introduce a new merging step where triples referring to adjacent sections of the reference sequence are merged together, as shown in Figure 5. This

merging can only occur for triples with no sym field (a null symbol), otherwise this information would be lost.

Variable length integers are used to encode the pos and len fields, whereby only the necessary amount of bytes needed to encode a value are used. The most significant bit in each byte is used as an end-of-integer flag, therefore a byte can encode 7 bits of the value. This optimisation is especially effective for the len field as the range of values can be very wide after merging.

Reference: AATGGGACGTGAGGGTTCCTCAGGCC

Sequence	Mapping triples $\langle pos, len, sym \rangle$
AATGGG	$\langle 0, 6, \_ \rangle$
ACGTGA	$\langle 6, 6, \_ \rangle$
GGGTTC	$\langle 12, 6, \_ \rangle$
Merge triples $\rightarrow$	$\langle 0, 18, \_ \rangle$

Fig. 5: Merging optimisation.

FASTQ format.

The FASTQ format is typically used to store the short reads output from NGS platforms. The reads are in no particular order, so merging cannot be exploited to improve the compression ratio. Since the reads could be sequenced from either strand of the DNA, it is possible to achieve a better mapping by using the reverse complement of the sequence. Observations indicate that for a typical sequencing data set, 30-40% of the sequences can be exactly matched to the reference sequence, however this percentage increases to 70-80% when additionally the reverse complement is considered. To improve the compression ratio we referentially compress both the sequence and its reverse complement, and choose the result with the fewest triples as the final encoding. Each sequence requires an additional bit in its encoding to determine if it is the original sequence or its reverse complement.

V. Hardware Design

In this section the hardware design for our mapping algorithm is presented. In addition, equations are derived for modelling the performance.

A. Design overview

Our reference-based compression tool targets computing systems with FPGA coprocessor boards. The host CPU reads in the to-be-compressed genomic data and offloads the sequences to the FPGA for reference-based compression. The results are transferred back to the CPU and written to disk.

The host CPU packages the input sequences into packets composed of: a sequence identifier, sequence length, and the sequence symbols (where each symbol is encoded using 2-bits). Sequences exceeding the packet allocation are split into smaller sub-sequences and compressed separately, after which the results are collated and the triples merged. The FPGA is configured with a module whose function is given by the mapping algorithm shown in Algorithm 2. Concisely written, each sequence is mapped to locations in the reference sequence, producing a set of triples. The module writes the output triples along with the associated sequence identifiers to off-chip DRAM attached to the FPGA device. After all the sequences have been mapped, the host reads the output triples directly from DRAM. The conversion of suffix array indices to reference sequence positions ($SA[low]$) is performed on the host CPU. In addition, the merging optimisation for the FASTA format is performed on the host CPU, as the data size is too small to justify FPGA acceleration. Figure 6 shows the hardware design overview. Note that the mapping module would be replicated as many times as possible according to the resources available on the FPGA device.

B. Module optimisation

The FM-index is stored in off-chip DRAM directly attached to the FPGA device. Accessing DRAM takes hundreds of cycles, which coupled with the step interdependence of the FM-index search operation, results in a non-filled pipeline of operations. To improve the module performance, the processing of multiple sequences is interleaved such that in each pipeline stage a different sequence is processed; consequently, the pipeline is completely filled, increasing the throughput. Furthermore, the DRAM memory controller is constantly processing commands, maximising the memory bandwidth utilisation. Interleaving is implemented using a circular buffer, where the buffer size is made equal to the total module latency. The trade-off is that additional logic and BRAM resources are required to store the batch of sequences and their corresponding mapping state. The additional resources can be minimised by reducing the module latency. For example, a binary adder tree is developed to count the occurrence in parallel.

In each cycle, two memory commands are sent to the memory controller requesting $F[low/d]$ and $F[high/d]$. A custom memory command generator is developed so that the

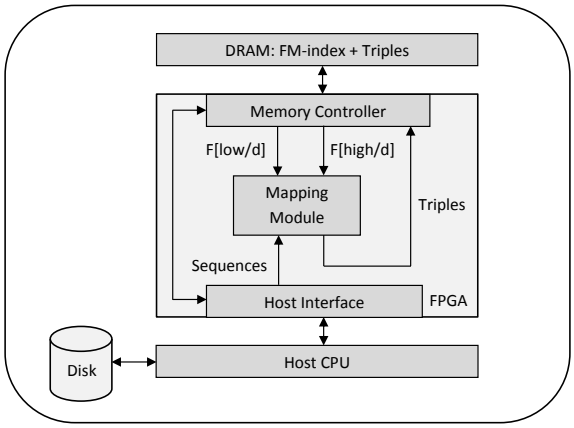

Fig. 6: Design overview. Note that arrows indicate data streams between the design components.

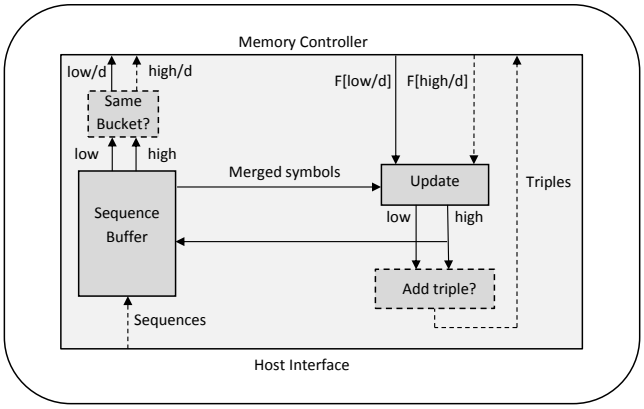

Fig. 7: Mapping module block diagram. Note that dashed lines indicate there is logic controlling the state of the stream.

index oversampling optimisation can be realised in hardware. For the $F[high/d]$ memory stream, a control bit is used to disable the command when low and $high$ point to the same bucket. In this case the bucket corresponding to $F[low/d]$ is used to update both low and $high$, reducing the number of memory commands processed. Multiplexers are used to select the appropriate input and computation result based on the values of low and $high$. Figure 7 shows a block diagram of the mapping module.

C. Performance modelling

The mapping time can be modelled as the maximum of a CPU component, and a DRAM component, as shown in Equation 1. The CPU component, T_{CPU}, covers the time spent on software tasks, such as generating the hardware input, and parsing the output. Since the FM-index search operation is memory bound, the DRAM component, T_{DRAM}, covers the hardware processing time. For large volumes of data, T_{DRAM} will dominate the compression time, whereas for small volumes, T_{CPU} will dominate.

$$T_{compress} = \max(T_{CPU}\ , T_{DRAM}) \tag{1}$$

The DRAM component can be modelled as the total number of bytes read from memory (B_r) divided by the DRAM bandwidth (BW_{DRAM}), as shown in Equation 2. Note that the number of bytes written to memory is omitted since it equates to less than 1% of the total DRAM usage.

$$T_{DRAM} = \frac{B_r}{BW_{DRAM}} \tag{2}$$

The DRAM bandwidth is platform specific, and depends on the number of memory channels available, the achievable bandwidth per channel, and the access pattern. The total number of bytes read from memory is calculated using Equation 3, where N_{access} is the total number of memory accesses, and B_{size} is the size of the FM-index buckets in bytes.

$$B_r = N_{access} \cdot B_{size} \tag{3}$$

The total number of memory accesses is well-characterised for the FM-index search operation. In the base algorithm, mapping a single symbol to the reference sequence requires two accesses to the index, therefore $N_{access} = 2 \cdot N_{seq} \cdot S_{len}$, where N_{seq} is the number of sequences to be compressed, and S_{len} is the number of symbols in the sequences. For the n-step FM-index, mapping n symbols to the reference sequence requires two accesses to the index; consequently, N_{access} is reduced by a factor of n. For index oversampling, after an average of X symbols have been matched to the reference sequence, a single access to the index is required to match each of the remaining symbols to the reference sequence, therefore $N_{access} = N_{seq} \cdot (2X + (S_{len} - X))$. The total number of index accesses when both the n-step FM-index and index oversampling are applied is calculated using Equation 4. The corresponding index bucket size in bytes is calculated using Equation 5, where d is the bucket size and n is the step size for the n-step FM-index. The summands in Equation 5 correspond to the occurrence markers and BWT section, where each symbol is encoded using 2 bits.

$$N_{access} = \frac{N_{seq} \cdot (2X + (S_{len} - X))}{n} \tag{4}$$

$$B_{size} = \frac{(32 \text{ bits} \cdot 4^n) + (2 \text{ bits} \cdot d \cdot n)}{8} \tag{5}$$

The equations derived allow the compression time to be accurately estimated for different data sets and hardware platforms; consequently, platform suitability can be assessed, and implementations verified.

VI. Evaluation

In this section we evaluate the performance of our compression tool implemented on a Maxeler MPC-X2000 dataflow node. Comparisons are made to other compression tools currently available.

A. Experimental setup

Our hardware design is implemented on a 1U Maxeler MPC-X2000 node [13] with 8 dataflow engines (DFEs). Each DFE comprises a single Altera Stratix V FPGA (28nm feature size) connected to 48GB of DRAM. The DFEs are connected to a CPU host machine via Infiniband. The design consumes 25% of the available slice registers, 27% of the LUTs, and 30% of the block RAMs on a Stratix V FPGA. The design runs at 200MHz, whilst the memory controller runs at the maximum 800MHz. The n-step FM-index is constructed with a bucket size $d = 128$, step size $n = 3$, and a sampling factor $f = 2$. These parameters are chosen so that the bucket width is the same as the DRAM burst width of the Maxeler Platform, therefore the available memory bandwidth is fully utilised. The FM-index size for the full Human genome is 17GB. If FPGA boards with smaller DRAM resources are targeted, the bucket size can be increased to reduce the index size.

Tool Name	Target Format	Notes
gdc2	FASTA	v2.0
fastqz	FASTQ	v15, mode e used
lw-fqzip	FASTQ	v1.01
fqzcomp	FASTQ	v4.6
gzip	General	v1.3.12
pbzip2	General	v1.1.6, 16 threads

TABLE I: Compression tools used in evaluation.

Each DFE supports a single channel to DRAM; consequently, no additional performance is observed when replicating the module as the DRAM bandwidth is already saturated. Furthermore, the memory controller is optimised for accessing large contiguous chunks of memory, rather than random access. Measurements indicate that the achievable memory bandwidth per FPGA is limited to 4.2GB/s, which is 11% of the theoretical peak. To improve the performance, the DRAM memory modules can be decoupled, allowing up to six memory channels per FPGA. This modification would permit up to two modules per DFE, and allow the FM-index buckets to be read in parallel. Although this modification has not been implemented yet, we use it to provide an upper bound performance estimate for the MPC-X2000.

The performance of our hardware-accelerated reference-based compression tool is compared to the compression tools listed in Table I. We also provide comparisons to a fully-optimised software version of our tool written in C++ (available on request). All the software tools are run on a 1U system with dual Intel Xeon X5650s (32nm feature size) and 120GB of DDR3-1333. 16 threads were used for tools supporting multi-threading. For this evaluation we compress the genomic data sets listed in Table II. The compression ratio is measured by dividing the sequence component of the original file and the sequence component of the compression output. Since this work only targets first order compression, we do not include second order compression in the compression time measurements. Disk I/O time is also not included for fairness, as the MPC-X2000 and CPU system have different storage devices. For our compression tool we use pbzip2 for second order compression.

B. Results

Tables III and IV show the compression statistics for the FASTA and FASTQ format data sets respectively.

File Name	Format	Seq. Size (MB)	Ref. Seq.
chr1 of hg19 [8]	FASTA	229	chr1
chr22 of hg19 [8]	FASTA	35	chr22
ERP001652 [6]*	FASTQ	964	hg19

TABLE II: Genomic data sets used in evaluation. *10 million reads are extracted from the ERP001652 data set.

	chr1		chr22	
Tool Name	C. Ratio	C. Time	C. Ratio	C. Time
gzip	3.2	55.1	3.1	8.6
pbzip2	3.7	3.12	3.8	1.2
gdc2*	10,556	1.1	10,547	0.17
Our Tool (CPU)	12,772	1.5	10,937	0.31
Our Tool (FPGA)	**12,772**	**0.54**	**10,937**	**0.08**

TABLE III: FASTA file compression. The compression time (C. Time) is measured in seconds. *gdc2 results are taken from [5], as the tool is more suited to compressing large batches of pre-assembled genomes.

For FASTA format compression we observe that our tool achieves a superior compression ratio and time compared to all the tools tested. For example, our tool achieves a 20% higher compression ratio, and is 2 times faster than gdc2 for the chr1 data set. The triple merging optimisation improves the compression ratio up to 41.3 times. This can be attributed to the high similarity between the to-be-compressed file and the reference sequence, such that the majority of the triples refer to adjacent sections of the reference sequence and can be merged. For the data sets tested, the compression time of our tool is dominated by software tasks, such as setting up the data streams; consequently, our tool only shows a small speed-up over the software tools.

For FASTQ format compression we observe that our tool achieves a superior compression ratio and time compared to all the tools tested. For example, our tool achieves a 30% higher compression ratio, and is 71.9 times faster than fastqz. The reverse complement mapping optimisation improves the compression ratio by 2 times. This can be attributed to the increase in exact mappings to the reference sequence, and that the best mapping is chosen from the original sequence and its reverse complement. For the data sets tested, the compression time of our tool is dominated by the hardware processing time; consequently our tool shows a more substantial speed-up over the software tools.

We observe that the n-step FM-index is the largest contributing factor to the hardware performance, providing a 3 times improvement over the base algorithm. The improvement from index oversampling is difficult to quantify as it depends on the reference sequence length. When the full Human genome is used as a reference sequence we observed that the performance improved by 1.8 times. We evaluate the performance modelling equations derived in Section V-C on the FASTQ format data, as the file is large enough that the compression time will be dominated by the hardware processing. The compression time is estimated to be 3.4 seconds, which is within 10% of the measured value. The main uncertainty in the estimation is the value of X for the index oversampling optimisation.

When comparing our hardware-accelerated tool to a fully-optimised software version, a less substantial speed-up is observed. The compression time improvement is approximately

Tool Name	C. Ratio	C. Time
gzip	3.3	252
pbzip2	3.7	11.0
fastqz	10.5	223
lw-fqzip	5.1	1567
fqzcomp	4.3	19.9
Our Tool (CPU)	14.1	10.4
Our Tool (FPGA)	**14.1**	**3.1**

TABLE IV: FASTQ file compression. The compression time (C. Time) is measured in seconds.

4 times over all the data sets tested. When normalising for the number of devices used, our hardware-accelerated tool is marginally faster than the software version of our tool, however we note that the hardware performance is currently limited by the MPC-X2000 memory architecture. When the upper bound estimate is considered, the performance improvement increases to 24 times over all the data sets tested. It can be argued that a mid-size CPU-based computational cluster could deliver the same performance as the MPCX-2000. However, this is not seen as an attractive solution: the form factor, power consumption and cooling requirements are not favourable for data centres and clinical settings. In contrast, the MPC-X2000 could be fully integrated with current NGS platforms, enabling on-the-fly compression of sequenced data as they are generated. For example, the FASTQ format compression speed for a single MPC-X2000 (312.5Mbps), is able to match the throughput of 44 Illumina HiSeq X platforms where each platform has a throughput of 6.9Mbps [9].

Table V shows the energy consumption for FASTQ format compression. The CPU power values are taken from the vendors' product information, whilst the FPGA device power is measured from the MaxOS operating system. The values in Table V indicate that our hardware-accelerated tool consumes approximately an order of magnitude less energy than the software version of our tool, and up to three orders of magnitude less energy than the other compression tools tested. This can be attributed to the low operational clock frequency, coupled with the short compression time. With relatively small energy consumption, form factor and cooling requirements, our tool is a promising candidate for integration in data centres and clinical settings.

Tool Name	Power (W)	Energy Consumption (kJ)
gzip	190	47.8
pbzip2	190	2.1
fastqz	190	42.4
lw-fqzip	190	297.8
fqzcomp	190	3.8
Our Tool (CPU)	190	2.0
Our Tool (FPGA)	**86**	**0.27**

TABLE V: Energy consumption for FASTQ format compression. Note that only device power is measured.

VII. Conclusion

This paper presents the first FPGA acceleration of reference-based compression for genomic data. We show that a hardware design based on a highly optimised version of the FM-index search operation can achieve good speed-up compared to software compression tools. Moreover, high compression ratios can be achieved without sacrificing performance. For FASTQ format compression, we observe that our tool achieves a 30% higher compression ratio and is 71.9 times faster than fastqz. Future work includes: applying memory channel optimisations, accelerating second order compression strategies, and involving additional genomic data sets in design evaluation.

Acknowledgement

We thank Peiyong Jiang, Dennis Lo and Rossa Chiu for their advice and encouragement. This work was supported in part by Maxeler University Programme, Altera, UK EPSRC Project EP/I012036/1, the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, and the HiPEAC NoE.

References

[1] J. Arram, W. Luk, and P. Jiang. Ramethy: Reconfigurable acceleration of bisulfite sequence alignment. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 250–259, 2015.

[2] J. K. Bonfield and M. V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PLoS ONE*, 8(3), 2013.

[3] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[4] A. Chacón et al. n-step FM-Index for Faster Pattern Matching . *Procedia Computer Science*, 18(0):70 – 79, 2013.

[5] S. Deorowicz, A. Danek, and M. Niemiec. Gdc 2: Compression of large collections of genomes. *arXiv preprint arXiv:1503.01624*, 2015.

[6] ERP001652 data set. http://www.ebi.ac.uk/ena/data/view/ERP001652.

[7] E. Fernandez et al. Multithreaded FPGA acceleration of DNA sequence mapping. In *Proc. High Performance Extreme Computing (HPEC)*, pages 1–6, Sept 2012.

[8] Human Genome version 19. http://hgdownload.cse.ucsc.edu/goldenpath/hg19/chromosomes.

[9] Illumina Hiseq X. http://www.illumina.com/systems/hiseq-x-sequencing-system.html.

[10] S. D. Kahn. On the future of genomic data. *Science(Washington)*, 331(6018):728–729, 2011.

[11] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990.

[12] T. Matsumoto, K. Sadakane, and H. Imai. Biological sequence compression algorithms. *Genome informatics*, 11:43–52, 2000.

[13] Maxeler Technologies. http://www.maxeler.com/products/mpc-xseries/.

[14] C. Olson et al. Hardware acceleration of short read mapping. In *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pages 161–168, April 2012.

[15] S. Wandelt and U. Leser. Fresco: Referential compression of highly similar sequences. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 10(5):1275–1288, 2013.

[16] Y. Zhang, L. Li, Y. Yang, X. Yang, S. He, and Z. Zhu. Light-weight reference-based compression of FASTQ data. *BMC bioinformatics*, 16(1):188, 2015.